Hachage de Descripteurs Locaux pour la Recherche d'Images Similaires

Adrien Auclair, Laurent D. Cohen, Nicole Vincent

Hachage de Descripteurs Locaux pour la Recherche d'Images Similaires

Local Descriptors Hashing for Near Duplicate Images Retrieval

Adrien Auclair[1] Laurent D. Cohen [2] Nicole Vincent[1]

[1] CRIP5 - Université Paris Descartes
[2] Ceremade - Université Paris Dauphine

{adrien.auclair,nicole.vincent}@math-info.univ-paris5.fr
cohen@ceremade.dauphine.fr

Résumé

Nous nous intéressons au problème de la recherche d'images similaires dans une large base de données. Les algorithmes les plus performants utilisent des descripteurs locaux d'images. Le problème devient alors d'utiliser un algorithme de recherche des plus proches voisins qui soit rapide. Dans cet article, nous proposons un nouvel algorithme basé sur une fonction de hachage originale. Notre algorithme sélectionne pour chaque point ses dimensions distinctives et s'en sert pour créer une clé de hachage. Nous comparons ses performances à plusieurs méthodes de l'état de l'art et montrons qu'il offre plusieurs avantages. Il est rapide, consomme peu de mémoire et ne demande pas d'apprentissage. Pour valider nos résultats, nous appliquons cet algorithme à une recherche d'images similaires dans une base de plus de 500.000 images. Enfin, nous montrons que notre algorithme est intégrable à une méthode de recherche par Bag-Of-Features.

Mots Clef

Recherche d'images similaires, descripteurs locaux, indexation.

Abstract

In this article, we study the problem of near-duplicate images retrieval within large scale databases. Using local image descriptors, this problem is linked to searching nearest neighbors of points in a high dimensional space. For this task, we propose a novel hash-based algorithm that detects distinctive dimensions for each point. These dimensions are used to compute hash keys. The obtained performances are compared with several state of the art algorithms. Our algorithm is fast, efficient in term of memory and does not require any learning stage. We successfuly applied it on a database of 500.000 images. Eventually, we show that our algorithm can efficiently be used in a Bag-Of-Features framework.

Keywords

Near duplicate images retrieval, local descriptors, indexing.

1 Introduction

L'application qui nous concerne dans cet article est la recherche d'images similaires dans des grandes bases d'images. Cela peut par exemple servir pour vérifier si une image vue sur Internet appartient à une banque d'images. Les tailles de ces banques d'images peuvent atteindre un million d'éléments. La difficulté vient du fait que l'image requête peut avoir subie une transformation par rapport à sa copie dans la base. Par exemple, il peut y avoir un recadrage, un lissage, de l'ajout de bruit, un redimensionnement... Pour être robuste à ces transformations, la classe d'algorithme qui s'est imposée dans l'état de l'art est celle utilisant des descripteurs locaux. Le plus efficace est appellé SIFT pour Scale Invariant Features Transform [7]. Dans cet article, nous utiliserons cet algorithme pour extraire des descripteurs des images étudiées.

Une fois ces descripteurs extraits des images de la base et de l'image requête, des correspondances sont cherchées entre ces deux nuages de points. Pour cela, un algorithme de recherche des plus proches voisins est utilisé. Dans une première partie de la littérature sur la recherche d'images similaires, toutes ces correspondances sont utilisées pour chercher une transformation affine entre l'image requête et des images de la base ([7, 6]. Pour être robuste aux outliers parmi les correspondances, un algorithme de type RANSAC [3] est utilisé pour estimer des matrices affines. Un des problèmes de ces méthodes est qu'elles sont difficiles à adapter aux très grandes bases à cause du temps nécessaire à l'étape de vérification affine. Pour s'adapter à ces bases, des auteurs ont proposé une approche dite par Bag-Of-Features [11]. L'idée de cette méthode est d'utiliser les correspondances trouvées pour calculer rapidement un score de similarité entre l'image requête et les images de la base. Les auteurs de [11] ont été inspirés par les algorithmes de

recherche de documents textuels similaires. Ils ont défini un vocabulaire visuel en appliquant une segmentation k-means au nuage des descripteurs de la base. Le score de similarité est ensuite calculé comme pour un document textuel en utilisant le modèle tf-idf (pour term-frequency inverse-document-frequency). Toutefois, pour obtenir une précision élevée, il est nécessaire d'appliquer l'étape de vérification affine avec les images qui obtiennent un bon score de similarité. L'avantage est que l'on peut limiter le nombre d'images à vérifier en ne choisissant que les plus similaires à la requête. Comme remarqué dans [5], l'utilisation du vocabulaire construit par k-means n'est en fait qu'un algorithme approché de recherche des plus proches voisins : deux descripteurs locaux associés au même mot sont des voisins.

Dans les deux classes d'algorithmes (vérification affine sur sur toutes les images ou filtrage par Bag-Of-Features), l'étape de recherche des correspondances est donc centrale. Il est alors crucial d'avoir un algorithme performant. Toutefois, les algorithmes existant sont gourmand en mémoire ou peuvent aussi nécessiter une longue phase d'apprentissage. Dans cet article, nous proposons un nouvel algorithme de recherche approchée des plus proches voisins, qui n'a pas besoin d'apprentissage, consomme peu de mémoire et offre de très bonnes perfomances.

2 Travaux antérieurs

Nous nous intéressons aux méthodes existantes pour effectuer une recherche de plus proches voisins. Les particularités de notre problème sont la taille des nuages de points ainsi que la grande dimension des espaces étudiés (128 dimensions dans notre cas). Ce dernier point est important car des algorithmes efficaces pour un espace à trois dimensions ne le seront pas forcément pour un espace à 128 dimensions (problème appellé "malédiction de la dimension"). Au dela de 10 dimensions, de nombreux algorithmes ne sont pas plus rapides qu'une recherche linéaire [12].

Pour pouvoir être plus rapide qu'une recherche linéaire, on trouve dans la littérature des algorithmes approchés de recherche des plus proches voisins. C'est-à-dire qu'on peut manquer quelques voisins, ou bien déclarer voisins des points qui ne le sont pas, mais en contrepartie, le temps d'execution est nettement réduit. Dans les paragraphes suivants, nous présentons les algorithmes qui ont inspiré nos travaux.

2.1 hachage LSH

L'algorithme LSH (pour Locality Sensitive Hashing) introduit dans [4] est un algorithme très utilisé dans ce cadre. Une fonction de hachage est dite "locality sensitive" si deux points proches obtiennent la même clé avec une forte probabilité et que deux points éloignés obtiennent la même clé avec une faible probabilité. Plus formellement, une famille de fonctions $\mathcal{H}\{h : S \mapsto U\}$ est sensible de type (r_1, r_2, p_1, p_2) avec $r_1 < r_2$ et $p_1 > p_2$ si on a les propriétés suivantes :

1. $\forall p \in B(q, r_1)$, alors $Pr_{h\in\mathcal{H}}[h(q) = h(p)] \geq p1$
2. $\forall p \notin B(q, r_2)$, alors $Pr_{h\in\mathcal{H}}[h(q) = h(p)] \leq p2$

où $B(q, r)$ est la boule de centre q et de rayon r.

On note g_i les l fonctions de hachage de la famille proposée dans [4] (i.e. les points sont stockés dans l tables de hachage différentes). L'algorithme est aussi paramétré par le nombre de dimensions qui seront hachées, noté k. Chaque fonction g_i est paramétrée par deux vecteurs :

$$D_i = \left\langle D_0^i, D_1^i, ..., D_{k-1}^i \right\rangle \tag{1}$$

et

$$T_i = \left\langle t_0^i, t_1^i, ..., t_{k-1}^i \right\rangle \tag{2}$$

Les valeurs de D_i sont choisies de manière aléatoire dans $[0, d-1]$ où d est le nombre de dimensions de l'espace. Les valeurs de T_i sont des seuils, choisis aléatoirement dans l'intervalle $[0, C]$, où C est la plus large coordonnée de tous les points du nuage. Chaque fonction g_i projette un point p de $[0, C]^d$ dans $[0, 2^k - 1]$ de telle manière que $g_i(p)$ est calculé comme une chaîne de k bits notés $b_0^i, b_1^i, ..., b_{k-1}^i$ tels que :

$$b_j^i = 0 \ \ si \ \left(p_{D_j^i} < t_j^i\right) \ sinon \ 1. \tag{3}$$

où l'on a noté $p_{D_j^i}$ la coordonnée de p selon la dimension D_j^i. La chaîne de k bits est la clé du point dans la i^{eme} table de hachage. Pour chercher les voisins d'un point requête q, on calcule sa clé de hachage pour chacune des tables. Ensuite, on applique une recherche linéaire sur les points des cases correspondantes. Les paramètres l et k permettent de choisir entre vitesse et précision.

Comme k peut être relativement élevé (e.g. $k = 32$), l'espace de destination des fonctions de hachage peut être trop grand pour tenir en mémoire. On rajoute alors une seconde fonction de hachage pour projeter le résultat des fonctions g_i sur un domaine plus petit. Toutefois, cela peut créer artificiellement des collisions. Pour détecter ces collisions, une somme de controle (aussi appellée checksum) est calculé sur chaque chaîne de k bits. Ce checksum permet d'identifier simplement si deux chaines de k bits sont différentes ou non. Lors du parcours linéaire des points d'une case, on ne calcule la distance que si ce checksum est identique pour le point en cours et le point requête. Cet algorithme a par exemple été utilisé avec succès dans [6].

Chaque table de hachage LSH peut-être vue comme un partitionnement aléatoire de l'espace. La clé de hachage obtenu pour un point est l'index de la partition à laquelle il appartient. L'inconvénient majeur du LSH est la quantité de mémoire requise car chaque point de la base est indexé dans un grand nombre de tables (i.e. l peut être grand). A noter qu'il existe une seconde version de l'algorithme LSH, introduite dans [1]. Toutefois, sur les nuages de points testés, nous n'avons pas constaté d'amélioration de performance significative avec la version présentée ici.

2.2 Segmentation par k-means

Dans [11], les auteurs créent un partitionnement du nuage de points avec un k-means. Les centres de classe peuvent ensuite être vus comme les graines d'un diagrame de Voronoï. L'algorithme de recherche des plus proches voisins est simple : deux points qui appartiennent à la même cellule de Voronoï sont déclarés voisins. L'étape la plus longue est d'assigner pour un point requête le centre de classe le plus proche (étape qui peut être accélérée avec l'approche hiérarchique de [9]). L'un des avantages de cette méthode est qu'elle nécessite beaucoup moins de mémoire que le LSH, chaque point de la base n'est indexé qu'une seule fois. En contrepartie, il faut appliquer un k-means au nuage de points étudiés. Cette phase est un problème car elle implique que si l'on change de base pour calculer ce vocabulaire, les performances peuvent s'avérer différentes.

2.3 Algorithme BOND

Nous présentons cet algorithme original [2] car il a en partie inspiré nos travaux. Il utilise des calculs de distances partielles pour accélérer une recherche exacte des k plus proches voisins. L'algorithme travaille dimension par dimension au lieu du parcours classique point par point. Les distances partielles sur un premier sous ensemble des dimensions sont calculées entre le point requête q et les points du nuage. La distance partielle du k^{ieme} point le plus proche permet d'éliminer les points qui sont déjà trop éloignés de q (selon la distance partielle). En itérant ce processus en rajoutant des dimensions dans la distance partielle, on diminue progressivement le nombre des voisins potentiels. L'algorithme est très lié au type de donnée et à l'ordre dans lequel sont utilisées les dimensions pour calculer les distances partielles. Il faut en effet trouver les dimensions pour calculer les distances partielles qui vont dès le début de l'algorithme éliminer un maximum de points. Cet algorithme montre l'importance d'analyser les données avant de concevoir un algorithme de recherche de plus proches voisins.

2.4 Positionnement

Nous avons cherché à développer un algorithme qui soit efficace en terme de consommation mémoire afin de pouvoir travailler sur une très grande base sans accès disque. C'est-à-dire que comme pour une segmentation k-means, un point de la base n'est indexé qu'une fois. Toutefois, un des inconvénients de cette méthode est que si k est grand, l'assignation d'une classe à un point peut être coûteuse en temps. On cherchera plutôt à construire une fonction de hachage rapide à calculer. Pour cela, nous nous sommes inspiré de l'algorithme BOND. Dans cet algorithme, les dimensions les plus distinctives de l'espace sont utilisées en premier pour calculer une distance partielle. Dans notre cas, nous cherchons, pour chaque point à créer une clé de hachage en fonction de ces dimensions distinctives. Une différence majeure est que nous proposons de chercher ces dimensions pour chaque point, et non de manière globale.

3 Hachage proposé

Pour chaque point de la base de données, ses k dimensions les plus distinctives sont détectées, avec k plus petit que le nombre total de dimensions (i.e. 128 pour les SIFT). Le vecteur de ces k dimensions est utilisé pour générer la clé de hachage de ces points. L'idée centrale est que deux points qui partagent les mêmes dimensions distinctives ont une bonne probabilité d'être proches. Intuitivement, un point est distinctif selon une dimension s'il est éloigné de la valeur moyenne selon cette dimension. Une autre idée est que les dimensions avec une forte variance sont les plus distinctives.
Pour traduire ces idées, nous introduisons les notations suivantes : $x^i = \langle x^i_1, x^i_2, ..., x^i_{128} \rangle$ le i^{eme} descripteur SIFT de la base de données et β une fonction qui mesure la distinctivité des descripteurs, par dimension :

$$\beta(x^i_j) = \left| \overline{x_j} - x^i_j \right| \sigma^\alpha_j \tag{4}$$

où $\overline{x_j}$ est la valeur moyenne des SIFT selon la dimension j , σ_j est l'écart type des SIFT selon cette même dimension et α est un paramètre. Cette fonction retourne une valeur élevée si une coordonnée est loin de la valeur moyenne selon une dimension avec un fort écart type. Le paramètre α permet de pondérer l'écart à la moyenne de ce point et l'écart type de cette dimension. Nous avons expérimenté plusieurs valeurs pour ce paramètre et $\alpha = 0.5$ donne les meilleurs résultats. Pour un point x^i, on note $D(x^i) = \langle D_1(x^i), D_2(x^i), ..., D_{128}(x^i)) \rangle$ le vecteur des dimensions (avec des valeurs dans $[1..128]$) triées par ordre décroissant de distinctivité :

$$\beta(x^i_{D_1(x^i)}) > \beta(x^i_{D_2(x^i)}) > ... > \beta(x^i_{D_{128}(x^i))} \tag{5}$$

Ainsi, $D_1(x^i)$ est la dimension selon laquelle le point x^i est le plus distinctif. Et $D_{128}(x^i)$ est la dimension selon laquelle le point x^i est le moins distinctif.
L'idée derrière notre structure d'indexation est que si deux points q et x^i sont proches, les premières valeurs des vecteurs $D(q)$ et $D(x^i)$ seront identiques (ou presque identiques, par exemple l'ordre peut varier). En d'autres termes, deux points proches sont distinctifs selon les mêmes dimensions. Pour un point x^i, les premières valeurs du vecteur $D(x^i)$ seront utilisées pour générer sa clé de hachage. Un exemple simple est d'utiliser les deux premières dimensions : les points de la base de données sont stockés dans un tableau à deux dimensions et le point x^i est stocké dans la case indexée par $(D_1(x^i), D_2(x^i))$. Ensuite, pour un point requête q, la recherche de ses voisins est limitée aux points de la case indexée par $(D_1(q), D_2(q))$.
Le problème de cette approche est que de nombreux voisins ne sont pas retrouvés car la fonction de hachage est trop sélective. Même si deux points x^i et q sont très proches, les premières valeurs de $D(x^i)$ et $D(q)$ ne sont pas forcément exactement les mêmes ou bien elles peuvent être ordonnées différemment. Cette difficulté est similaire à celle

expliquée dans [4] lors de l'utilisation d'une seule fonction LSH. Leur solution est d'utiliser plusieurs fonctions de hachage, chacune associée à une table. Comme nous l'avons dit précédemment, cette solution a l'inconvénient de consommer beaucoup de mémoire. Dans notre solution, un point de la base est haché dans une seule case de la table mais plusieurs cases sont parcourues lors de la recherche des voisins (voir 3.1 pour cet algorithme de recherche). Afin d'éviter que deux vecteurs de k dimensions contenant les mêmes valeurs mais dans deux ordres différents soient hachés dans deux cases différentes, les valeurs des vecteurs $D(x^i)$ sont triées par ordre croissant avant de générer la fonction de hachage.
La table de hachage finale notée H est un tableau à k dimensions. Chaque dimension du tableau est indexée par un entier entre 1 et 128. Pour un point x^i, les k premières valeurs de $D(x^i)$, triées par ordre croissant forment un vecteur noté : $tri(D(x^i)) = \langle a_1, a_2, ..., a_k \rangle$ avec

$$1 \leq a_1 < a_2 < ... < a_k \leq 128 \tag{6}$$

et ainsi, le point x^i est stocké dans la case $H[a_1][a_2]...[a_k]$. En pratique, pour des valeurs de k pas trop petites (e.g. k supérieur à 5), la table requiert beaucoup trop d'espace mémoire. Pour cette raison, on utilise une seconde fonction de hachage comme dans [4]. On appelle g cette fonction qui projette un vecteur à k dimensions sur un unique entier dans l'intervalle $[0, c]$. On utilise pour cela une fonction g du type donné dans [10] :

$$g(tri(D(x^i))) = \left(\left(\sum_{i=1}^{k} r_i a_i \right) mod\, P \right) mod\, c \tag{7}$$

où P est un nombre premier et r_i sont des entiers aléatoires. Ainsi, la table de hachage n'a plus qu'une dimension. Un point x^i est stocké dans cette nouvelle table de hachage H' à : $H'[g(tri(D(x^i)))]$.
Cette fonction g introduit de nouvelles collisions qui n'auraient pas eu lieu avec la table H. Pour les détecter, nous utilisons une seconde fonction de hachage similaire à g pour calculer une somme de controle (dite checksum) à partir de vecteurs à k dimensions. Un point est alors stocké dans la table sous forme d'une référence (i.e. son id) et du checksum associé.

3.1 Algorithme de recherche

Pour un point requête q, nous cherchons à savoir quelles cases de la table on doit parcourir pour retrouver avec une forte probabilité ses voisins. Pour cela, le vecteur de ses dimensions les plus distinctives $D(q)$ est calculé. Toutes les combinaisons de k valeurs (triées par valeurs croissantes) parmi les n premières sont sélectionnées. Chaque combinaison est hachée et les cases de la table sont parcourues linéairement pour trouver les voisins. Lors du parcours d'une case, les distances ne sont calculées qu'avec les points qui ont un checksum identique à celui de la combinaison testée et les autres sont ignorés. Cet algorithme impose de parcourir C_n^k cases de la table de hachage. Un élément très important est la faible consommation mémoire de cet algorithme. En effet, un point de la base est indexé dans une seule case de la table.

4 Evaluation

Dans une première partie, nous évaluons l'algorithme proposé uniquement du point de vue de la recherche des plus proches voisins. Dans ces tests, nous considérons que deux descripteurs SIFT sont voisins si la distance euclidienne entre ces points est inférieure à 250 (seuil déterminé expérimentalement sur notre implémentation pour obtenir un très bon rappel dans une recherche d'images similaires).
Dans ce travail, nous cherchons à obtenir un algorithme qui retourne le plus rapidement possible une liste approchée des voisins recherchés. Pour mesurer la rapidité des algorithmes de manière indépendante de l'implémentation, nous considèrons ces algorithmes comme des algorithmes de filtrage. C'est-à-dire qu'au lieu de calculer les distances euclidiennes avec tous les points du nuage, un algorithme sélectionne un sous ensemble de ce nuage et ne calcule les distances que sur ce sous ensemble. Le temps de recherche des voisins d'un point requête q est alors égal à :

$$T(q) = T_f(q) + T_l(q) \tag{8}$$

où $T_f(q)$ est le temps nécessaire pour filtrer les points du nuage. Par exemple, si la méthode est basée sur un hachage, ce temps sera celui nécessaire pour calculer la fonction de hachage pour le point q. Ensuite, $T_l(q)$ est le temps nécessaire pour calculer les distances euclidiennes entre le point q et tous les points du sous ensemble filtré. Le temps $T_f(q)$ est constant et le temps $T_l(q)$ est linéaire par rapport au nombre de points retournés, c'est donc ce dernier qui nous intéresse pour les grands nuages de points que nous traitons. Au lieu de mesurer le temps d'exécution de ces algorithmes pour les comparer, il est plus intéressant de mesurer le ratio entre ce nombre de points filtrés et le nombre de points du nuage initial (car indépendant de l'implémentation). Si le nuage de points est assez grand, on a $T_f(q) << T_l(q)$ et le temps de calcul est alors linéairement proportionnel au nombre de points dans le sous ensemble filtré. Nous noterons $RatioFilter$ ce ratio entre le nombre de points du sous-ensemble et le nombre de points du nuage initial.
Au final, pour évaluer les algorithmes, nous mesurerons le rappel obtenu ainsi que $RatioFilter$. Un algorithme sera d'autant plus efficace qu'il aura un rappel proche de 1 pour un $RatioFilter$ proche de zéro.
La figure 4 montre les courbes $RatioFilter$/Rappel pour plusieurs algorithmes. On remarque que pour un $RatioFilter$ donné, notre algorithme a un rappel plus élevé que les deux autres algorithmes testé. Par exemple, pour un $RatioFilter$ de 0.01, le hachage proposé retrouvera environ 85% des voisins alors que les autres algorithmes en trouveront 70%.

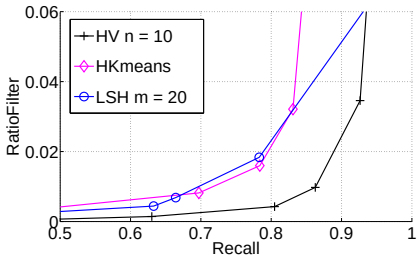

FIGURE 1 – Comparaison de l'algorithme proposé, noté HV avec d'autres algorithmes pour un seuil de distance de 250 sur un nuage de 265.000 points. HKmeans correspond à une segmentation k-means (le nombre de classes utilisées varie le long de la courbe). Pour le LSH, le nombre de tables est 20 et le nombre de dimensions sélectionnées pour calculer les clés de hachage varie sur la courbe.

Dans la partie suivante, nous cherchons à confirmer ces résultats pour l'application de recherche d'images similaires.

5 Recherche d'images similaires

Pour ces tests, nous utiliserons trois bases d'images afin de valider notre algorithme sur des bases de taille variable. La première base notée DB_{4000} est constituée à partir de deux jeux d'images. Le premier jeu est construit à partir de 50 images téléchargées depuis Internet. Ces 50 images correspondent aux 50 requêtes utilisées pour tester l'algorithme. Nous appliquons à chacune de ces images 53 transformations (détaillées dans le paragraphe suivant). Nous avons alors 2650 images (50×53) à retrouver dans notre jeu de données. Nous ajoutons ensuite 1350 images extraites de la base "holiday" de l'INRIA. Pour mesurer le rappel et la précision, nous effectuons les 50 requêtes, et pour chacune d'elle, nous comptons le nombre d'images retrouvées parmi les 53. Nous pouvons aussi mesurer le rappel et la précision pour chaque transformation.
La seconde base est constituée de DB_{4000} à laquelle on rajoute 28.000 images de la base "holiday". On appelle cette base DB_{32k}. Enfin, la dernière base contient 510.000 images. Ces images sont issues de plusieurs collections. La base "holiday" de 100.000 images est incluse. Les 2650 images à retrouver dans nos tests sont aussi incluses. Le reste des images provient de divers sites internet.
La base DB_{4000}, de petite taille permet de tester rapidement de nombreux couples de paramètres. La seconde permet de confirmer ces résultats sur une base de taille moyenne. Enfin, la base de 510.000 images permet de valider notre algorithme sur une très large base.
La liste des transformations que nous utilisons est la suivante (nous donnons un nom unique à chaque transformation, et le nombre entre parenthèses est le nombre d'images transformées générées) :

1. COULEUR (3) : Augmenter la composante (a) rouge, (b) verte ou (c) bleue de 10%.
2. CONTRASTE (4) : (a) Augmenter ou (b) diminuer le contraste selon le paramètre par défaut, (c) augmenter (d) diminuer le contraste en utilisant trois fois le paramètre par défaut.
3. CROP (5) : Recadrer l'image en supprimant (a) 10%,(b) 25%, (c) 50%,(d) 75%, ou (e) 90% de l'image (en conservant la zone centrée autour du centre de l'image). L'image est ensuite redimensionnée à la taille de l'image initiale.
4. DESPECKLE (1) : utilisation de l'opérateur despeckle de ImageMagick.
5. GIF (1) : Convertir l'image d'entrée au format GIF. Cela compresse l'espace des couleurs sur 8 bits.
6. ROTATION (3) : Tourner l'image par (a) $10°$, (b) $45°$ ou (c) $90°$.
7. REDIM (6) : Redimensionner l'image pour que sa taille soit multipliée par (a) 2, (b) 4, (c) 8 times ou diviser par (d) 2, (e) 4, (f) 8.
8. SATURATION (5) : Changer la saturation de l'image par (a) 70%, (b) 80%,(c) 90%, (d) 110% ou (e) 120%.
9. INTENSITY (6) : Changer l'intensité de l'image par (a) 50%, (b) 80%,(c) 90%, (d) 110%,(e) 120%, (f) 150%.
10. NETETE (2) : Opérateur Sharpen de ImageMagick avec paramètre (a) 1 et (b) 3.
11. EMBOSS (1) : Opérateur Emboss de ImageMagick, paramètre à 1.
12. JPEG (2) : Fixer la qualité JPEG à (a) 80 et (b) 15.
13. MEDIAN (1) : filtre median 3x3.
14. ROTCROP (2) : Tourner l'image de (a) $10°$ et (b) $45°$ et recadrer pour ne conserver que 50% de l'image.
15. ROTSCALE (2) : Transformation ROTCROP et REDIM pour diviser la taille de l'image par 4.
16. INCLINAISON (4) : Incliner l'image selon l'axe x de (a) $5°$, (b)selon l'axe x et y de $5°$, (c) selon l'axe x de $15°$ et selon les deux axes de $15°$.
17. GAUSS (3) : Flou gaussien de rayon (a) 3, (b) 5 et (c) 7.
18. GAUSSNOISE (2) : Ajout de bruit gaussien d'écart type (a) 10 et (b) 20.

La figure 2 montre quelques unes des images transformées parmi les plus difficiles à retrouver.
Les performances de plusieurs algorithmes sont montrés sur le tableau 3. Sur les deux premières lignes, nous avons mis les résultats de la recherche linéaire sur CPU (processeur à 2.66GHz) et sur GPU (Nvidia 8800 GT). L'implémentation sur GPU est 35 fois plus rapide que la version CPU. Avec cette version de l'algorithme (recherche exacte), seules 14 images sont manquées. Pour comparer au LSH, nous avons mis dans le tableau les deux couples de paramètres qui donnent un rappel d'environ 0.98. Pour cette valeur, le hachage proposé avec

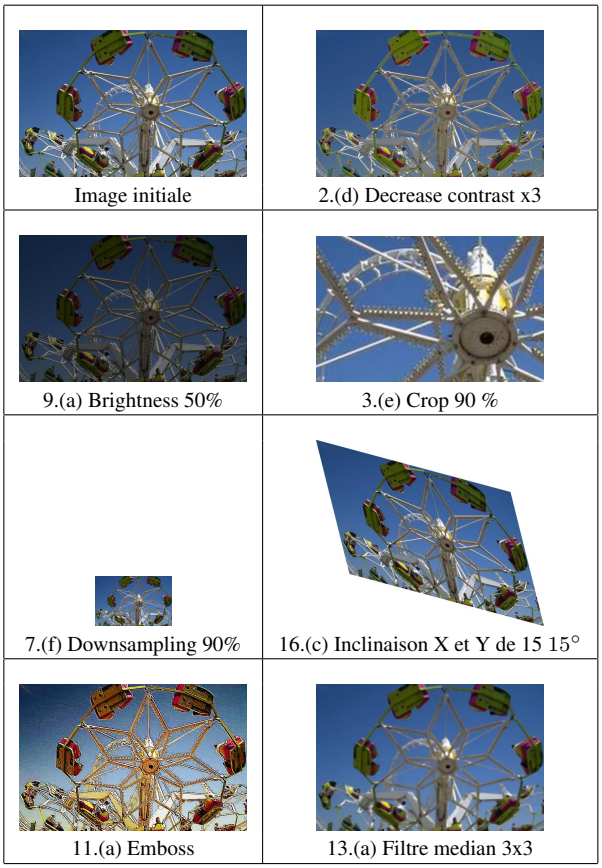

FIGURE 2 – Exemples des transformations d'images utilisées.

$n, k = 8, 6$ est environ 4 fois plus rapide que le LSH. D'autre part, il faut garder en tête qu'avec ces paramètres notre hachage consomme 20 fois moins de mémoire que le hachage LSH. On remarque aussi que la plupart des images non retrouvées sont issues des transformations $INCLINAISON15\_15$,CROP90 et EMBOSS. Si nous acceptons de ne pas retrouver les images de ce type, en paramétrant notre algorithme avec $n, k = 8, 6$ nous obtenons un temps de recherche des plus proches voisins de 62 ms, pour un rappel de 0.967.

Sur la base DB_{32k}, nous testons notre algorithme avec $n, k = 12, 10$. Nous obtenons le même rappel et la même précision que sur le base DB_{4000}. Concernant le rappel, il est normal d'obtenir le même résultat car la taille de la base n'affecte pas les voisins SIFT retrouvés. Nous pourrions toutefois nous attendre à ce qu'en rajoutant des images, la précision diminue, ce qui n'est pas le cas. Cela confirme que la méthode de vérification affine permet de ne conserver que les images qui sont vraiment similaires. Au niveau des performances en temps, le temps de recherche des voisins SIFT est de 187ms. On voit que ce temp augmente peu par rapport au temps obtenu pour la petite base (i.e. 62 ms).

Pour notre base de 510.000 images, si l'on fixe un maximum de 256 SIFT par image, il faut alors $510.000 \times 256 \times 144 = 18.8Go$ pour stocker les descripteurs. Une telle quantité de donnée de tiens pas en RAM. Avec l'algorithme tel que décrit dans les paragraphes précédents, cela impose de charger sur le disque les descripteurs SIFT, à la demande, pour calculer la distance euclidienne et vérifier si ce sont bien des voisins du point requête. Nous proposons donc d'ignorer cette étape. C'est-à-dire que nous ne filtrons pas les points en fonctions de leur distance au point requête. L'inconvénient est que nous allons accepter des points qui ne sont pas de vrais voisins, mais en contrepartie, nous pouvons ne pas charger en mémoire les descripteurs SIFT.

Pour ces tests, nous fixons un maximum de 256 SIFT par image. Sur une machine à processeur Intel QuadCore Xeon 2.66GHz, en traitant plusieurs images en parallèle, le temps moyen de calcul des descripteurs est de 0.34s par image. Le temps total sur les 510.000 images est d'environ 48 heures. Cela génère 128 million de points, qui occupent 18Go sur le disque.

Notre implémentation utilise plusieurs tables pour stocker le système d'indexation. Une première table liste les images présentes en base. Ensuite, nous enregistrons dans une table tous les points d'intérêts des descripteurs SIFT de la base. C'est-à-dire que comme notre méthode n'utilise pas le descripteur lui-même (le vecteur à 128 dimensions), mais seulement sa position, son orientation et son échelle, seuls ces 4 paramètres sont enregistrés. Pour optimiser l'espace mémoire nécessaire, ces 4 valeurs sont codées sur un total de 64 bits (16 bits pour chaque champ). Pour notre base, cette table occupe 1.02 Go. Enfin, nous enregistrons aussi la table de hachage dans un fichier. Chaque élément de la table est une structure de 64 bits. Les 24 premiers bits enregistrent l'id de l'image auquel fait référence cet élément. Les 8 bits suivants codent l'index du descripteur parmi les descripteurs de cette image. Enfin, les 32 derniers bits permettent de stocker le checksum de cet élément. Cette table occupe aussi 1Go pour la base de 510.000 images. La construction de cette indexation (avec $n, k = 10, 12$) prend environ deux heures sur notre machine.

Avec cette implémentation, il faut une minute pour que le moteur de recherche charge ces tables. Cela requiert environ 2Go de RAM. Nous pouvons ensuite executer des recherches sans effectuer aucun accès disque.

Nous reprenons le protocole de test décrit précédement (pour lequel 50 requêtes sont effectuées et pour chacune d'elles, les 53 images déformées doivent être retrouvées). Le temps moyen de recherche des descripteurs voisins est de 248ms. Le temps de vérification affine est de 9980ms par image. Nous obtenons un rappel de 0.966 et une précision de 0.970.

Il faut comparer ces valeurs à celle obtenues pour le test où nous vérifiions la distance euclidienne entre les descripteurs. Dans la table 3, nous avions utilisé un seuil de 200. Nous obtenions, sur une base de 4000 images, pour $n, k = 10, 12$ un rappel de 0.967 et une précision de 1.

Sur la très large base, le rappel est quasiment identique. Par

Algo	params	R	P	TNN (ms)	$INCLINAISON15\_15$	CROP90	EMBOSS	MEDIAN3	INCLINAISON15	CROP75
Linéaire	sur CPU	0.995	1	167418	9	4	1			
Linéaire	sur GPU	0.995	1	4756	9	4	1			
HV	n,k=12,10	0.967	1	62	44	28	14		1	1
HV	n,k=8,6	0.980	1	129	31	15	5	1		
LSH	m,k=20,140	0.978	1	508	36	17	7	1		

FIGURE 3 – Résultats de l'application de recherche d'images similaires pour plusieurs algorithmes de recherche des plus proches voisins.

contre, la précision est inférieure. Il y a deux raisons à cela. D'abord, en ne vérifiant pas la distance, nous acceptons plus de fausses correspondances, et donc nous pouvons retrouver des images alors qu' il ne faudrait pas. Ensuite, la base est beaucoup plus grande, et la probabilité qu'il y ait des images réellement similaires aux images requêtes est plus élevée. En vérifiant manuellement, nous nous aperçevons que cette seconde raison est la source principale des fausses erreurs. En téléchargeant aléatoirement des images sur internet, certaines l'ont été plusieurs fois. Si c'est le cas pour une image utilisée comme requête dans notre test, nous allons retrouver les 53 images modifiées, mais aussi une copie de la requête téléchargée sur un autre site. En pratique il s'agit là de la plupart des cas. Il faudrait supprimer ces images de la base pour obtenir une meilleure mesure de la précision.

Après avoir montré que notre algorithme s'intégrait bien dans une application classique de recherche d'images similaires, nous montrons dans les paragraphes suivant qu'il est aussi possible de l'intégrer dans une méthode Bag-Of-Features.

5.1 Intégration à une méthode Bag-Of-Features

Comme vu dans l'introduction, un algorithme de recherche des plus proches voisins peut aussi être intégré dans une méthode par Bag-Of-Features. Avec notre hachage, cela revient à assigner un mot à chaque case de la table de hachage.

Dans [5], les auteurs montrent que le score de similarité entre l'image requête et la i^{eme} image de la base peut être exprimé sous la forme :

$$s_j = \sum_{i=1}^{n} \sum_{k=1}^{m_j} f(x_i, p_k^j) \tag{9}$$

où m_j est le nombre de descripteur dans la j^{eme} image, n est le nombre de descripteur dans l'image requête et $f(x, y)$ est une fonction de coût qui exprime la ressemblance entre deux descripteurs. Dans l'algorithme initial de [11], les auteurs utilisaient (par souci de simplicité, on n'a pas intégré ici la pondération if-tdf) :

$$f(x, y) = \delta_{q(x),q(y)} \tag{10}$$

où x est un descripteur de l'image requête, y un descripteur de la base (appartenant à l'image j), $\delta_{a,b}$ est l'opérateur de kronecker (i.e. qui vaut 1 si $a = b$ et 0 sinon) et q est une fonction qui donne l'index du mot auquel appartient un descripteur SIFT (i.e. le centre des k classes le plus proche du point).

Dans cet algorithme, nous remplacons cette fonction par :

$$f'(x, y) = \frac{1}{n}\frac{1}{n_j} \times \delta'_{hash(x),hash(y)} \times log(N/N_c)^2 \tag{11}$$

où x est un descripteur de l'image requête, y un descripteur de la base (appartenant à l'image j), n le nombre de descripteurs de l'image requête, n_j le nombre de descripteurs de l'image j, N le nombre total de descripteurs dans la base, N_c le nombre de descripteurs de la base qui sont hachés dans la table en $hash(y)$ et où la fonction δ' est très similaire à la fonction de kronecker mais adaptée pour notre cas (voir formule 12). Le terme $\frac{1}{n}\frac{1}{n_j} \times \delta'_{hash(x),hash(y)}$ correspond à la partie term-frequency du schéma tf-idf alors que la partie $log(N/N_c)^2$ correspond au terme inverse-document-frequency.

Dans la formule 11, y est un point de la base et est donc haché à une unique case de la table. Inversement, x est un point requête et il est donc haché dans plusieurs cases de la table (i.e. $hash(x)$ est une liste de clé alors que $hash(y)$ est une clé unique). Nous définissons alors :

$$\delta'_{hash(x),hash(y)} = \begin{cases} 1 & \text{si } hash(y) \in hash(x) \\ 0 & \text{sinon} \end{cases} \tag{12}$$

Pour évaluer cet algorithme, nous reprenons le protocole de test décrit dans les paragraphes précédents. Nous n'utilisons que la base DB_{32k}. L'algorithme retourne uniquement les 53 images qui obtiennent les meilleurs scores de similarité avec l'image requête. Aucune vérification affine n'est effectuée. Les résultats sont présentées dans la table 1. On voit que sur cette base, le hachage proposé obtient un meilleur rappel que la méthode du vocabulaire par k-means tout en étant plus rapide. A noter que dans ces tests, le vocabulaire k-means a été construit sur une autre base que celle dans laquelle la recherche est effectuée. Or il est connu que les meilleurs résultats sont obtenus pour un vocabulaire calculé sur la base de recherche. On relance alors ce test, en changeant aussi la base sur laquelle est calculée le vocabulaire. Les résultats sont présentés sur la table 2 (pour des raisons de rapidité, nous avons utilisé la base

Algo	R	manquées	T(ms)
n,k=10,8	0.970	102	29
$kmeans$,k=4000	0.900	266	275
$kmeans$,k=16000	0.927	192	310
$kmeans$,k=32000	0.935	172	385
$kmeans$,k=64000	0.943	151	593

TABLE 1 – Rappels obtenus par une recherche Bag-Of-Features utilisant notre hachage et un vocabulaire k-means, sur la base DB_{32k}. Seules les couples de paramètres les plus performants sont présentés ici. Pour la méthode k-means, le vocabulaire est calculé sur une autre base.

Algo	R	manquées
n,k=12,10	0.969	82
$kmeans^1$,k=16000	0.993	19
$kmeans^2$,k=16000	0.967	88

TABLE 2 – Images manquées par une recherche Bag-Of-Features utilisant notre hachage et un vocabulaire k-means, sur une base de 4000 images. (1) : le vocabulaire est calculé sur la base de tests. (2) :le vocabulaire est calculé sur une autre base.

DB_{4000} pour ce test). On remarque alors que si le vocabulaire est calculé sur la base de recherche, la méthode par k-means obtient le meilleur rappel.

6 Conclusion

Dans cet article, nous avons proposé un nouvel algorithme de recherche approchée des plus proches voisins. Cet algorithme est basé sur une fonction de hachage qui utilise les indices des dimensions les plus distinctives de chaque point pour calculer sa clé. Pour être efficace en mémoire, les points de la base sont hachés dans une seule case de la table. Inversement, pour un point requête, plusieurs clés sont calculées pour augmenter la probabilité de retrouver des points voisins. Nous avons montré que sur des points SIFT, notre algorithme obtient de meilleurs résultats que le hachage LSH ou bien qu'une recherche utilisant une segmentation k-means. Pour l'application de la recherche d'images similaires, nous avons d'abord testé notre algorithme en le couplant avec une vérification affine. Dans ce cadre, il est plus intéressant que le LSH, notamment grâce à sa faible consommation mémoire. Nous avons aussi montré qu'il permet de gérer, sans accès disque, une base de 510.000 images. Il s'intègre aussi simplement dans une méthode par Bag-Of-Features. Dans ce cas, il obtient de meilleurs résultats qu'un vocabulaire par k-means (si celuici n'est pas calculé sur la base de recherche). Dans nos travaux futurs, il sera intéressant de comparer les performances de notre algorithme à celles obtenues dans les travaux de [5] et aussi celles des algorithmes de randomized forest [8].

Références

[1] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SCG '04 : Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, New York, NY, USA, 2004. ACM.

[2] A. P. de Vries, N. Mamoulis, N. Nes, and M. Kersten. Efficient k-nn search on vertically decomposed data. In *SIGMOD '02 : Proceedings of the 2002 ACM SIGMOD international conference on Management of data, Madison, USA*, pages 322–333, 2002.

[3] M. A. Fischler and R. C. Bolles. Random sample consensus : A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6) :381–395, 1981.

[4] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *The VLDB Journal*, pages 518–529, 1999.

[5] H. Jegou, M. Douze, and C. Schmid. Hamming embedding and weak geometric consistency for large scale image search. In *Proceedings of the European Conference on Computer Vision (ECCV), Marseille, France*, LNCS. Springer, oct 2008.

[6] Y. Ke, R. Sukthankar, and L. Huston. An efficient parts-based near-duplicate and sub-image retrieval system. In *MULTIMEDIA '04 : Proceedings of the 12th annual ACM international conference on Multimedia*, pages 869–876, New York, NY, USA, 2004. ACM Press.

[7] D. G. Lowe. Distinctive image features from scale-invariant keypoints. In *International Journal of Computer Vision (IJCV)*, volume 20, pages 91–110, 2004.

[8] F. Moosmann, E. Nowak, and F. Jurie. Randomized clustering forests for image classification. *IEEE Transactions Pattern Analysis and Machine Intelligence (PAMI)*, 30, September 2008.

[9] D. Nistér and H. Stewénius. Scalable recognition with a vocabulary tree. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), New York, USA*, pages 2161–2168, 2006.

[10] G. Shakhnarovich, T. Darrell, and P. Indyk. *Nearest-Neighbor Methods in Learning and Vision : Theory and Practice (Neural Information Processing)*. The MIT Press, 2006.

[11] J. Sivic and A. Zisserman. Video Google : A text retrieval approach to object matching in videos. In *Proceedings of the International Conference on Computer Vision (ICCV), Nice, France*, volume 2, pages 1470–1477, October 2003.

[12] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB, New York, USA*, pages 194–205, 24–27 1998.